Projeto e Análise de Algoritmos - Trabalho Prático 1: Grafos - Problema Político

Lucas de Miranda Bastos[1]

[1]Departamento de Ciência da Computação - Universidade Federal de Minas Gerais

1. Introdução

Atualmente, mais de 60 países possuem o regime presidencialista total [1], nesses sistemas, um chefe de estado, comumente chamada de *presidente*, lidera o poder executivo. Contudo, nessa mesma forma de governo, o presidente precisa de alguns aliados, tando do executivo, quanto do legislativo, para que suas políticas sejam fortalecidas e seus projetos tenham grandes chances de serem aprovados.

Conseguir apoio de aliados, não é uma tarefa simples, tomando como exemplo uma câmara de deputados, existe uma rede complexa de similaridade entre os componentes, formadas devido à associação partidária, interesses pessoais ou interesses políticos em comum. De forma simples: o presidente deve tentar escolher o menor número que deputados, mas ainda maximizem o número de outros deputados que possam ser indiretamente aliados através da similaridade. A razão de focar em poucos deputados se deve ao fato de que uma aliança normalmente envolve movimentações de verbas lícitas e ilícitas, abertura de cargos públicos e outras medidas que podem colocar em risco a reputação do presidente. Quando um deputado possui similaridade com muitos outros deputados direta ou indiretamente, é dito que o mesmo possui uma alta **capacidade de intermediação**.

Sendo possível parametrizar a similaridade entre deputados, o problema pode ser abstraído em um problema utilizando-se grafos, onde os vértices representam os deputados e as arestas representam a similaridade entre os mesmos (Figura 1). A saída consiste em uma lista ordenada em ordem decrescente de sua capacidade de intermediação. Desta forma, o presidente possuirá uma informação preciosa para ajudar a decidir quais deputados escolher. (Para este problema, em caso de empate entre dois ou mais deputados, aqueles que possuírem menor id devem aparecer primeiro na lista).

1.1. Modelagem e Resolução do Problema

Uma vez que o problema foi modelado como um problema de grafos, pode-se utilizar todo um arcabouço de algoritmos para resolvê-lo. Mas para que isso seja possível, primeiramente é necessário definir como a capacidade de intermediação de um deputado x pode ser calculada. Para ilustrar todo o raciocínio será utilizada a figura 1 como exemplo.

Primeiramente, note que um deputado não pode ter relações diretamente com aqueles deputados com quem não tem similaridade, no exemplo, o deputado 3 não pode se relacionar diretamente com o deputado 2, ele precisa da intermediação do deputado 1 para fazer a comunicação. Considerando apenas esses deputados (1,2,3), pode-se dizer que **o deputado com a a maior capacidade de intermediação é o deputado 1, uma vez que o mesmo é necessário na relação entre 2 e 3**. De forma similar, temos a situação onde os deputados 1 e 3 são fundamentais para a comunicação entre 2 e 4 (Definição 1).

[1]`https://pt.wikipedia.org/wiki/Lista_de_pases_por_sistema_de_governo`

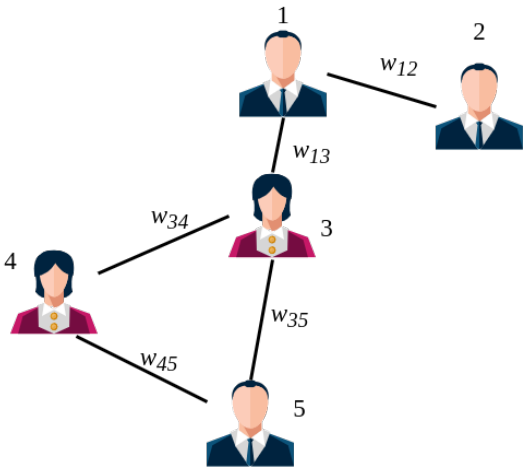

Figura 1. Ilustração da Rede de Similaridade entre deputados, modelada como um grafo. O deputado 3 possui certa similaridade com todos os outros 4 deputados, portanto parece ser um forte candidato a ter maior capacidade de intermediação. Note que se w_{35} possuir um valor muito baixo em relação a w_{34} e w_{45}, é melhor contar com a intermediação do deputado 4 em relações do deputado 3 com o deputado 5.

Definição 1. *A capacidade de intermediação de um deputado, é dada pelo número de relações que o mesmo faz a intermediação.*

Entretanto, podem existir vários caminhos entre dois deputados, no exemplo, o deputado 3 pode se relacionar diretamente com o deputado 5 ou utilizar a intermediação do resultado 4, sendo assim, é necessário escolher a melhor opção. O ponto chave desse problema se encontra nos pesos das arestas. Uma vez que as arestas mapeiam a similaridade entre deputados, um bom caminho é aquele que passa por arestas com alta similaridade. (Uma relação entre dois deputados com 70% de similaridade é preferível sobre uma relação entre deputados com apenas 20% de similaridade).

Sendo assim, o melhor caminho é aquele cujo somatório dos pesos das arestas é o maior possível. Este problema é similar ao problema do caminho mínimo, mas ao invés de procurar pelas menores arestas, ele procura pelas maiores. Portanto, para trabalhar com algoritmos de caminho mínimo, é necessário fazer algum tipo de normalização das arestas. Considerando que as similaridades (s) são dadas em porcentagem ($0 \leq s \leq 100$), pode-se utilizar o complemento para normalizar as arestas:

$$w' = 100 - w \tag{1}$$

Onde w' é o novo peso da aresta e w é o antigo peso das arestas.

Com isso é possível reescrever a definição 1 a fim de se obter a solução para o problema (Definição 2).

Definição 2. *Dado um grafo G tal que as arestas foram normalizadas segundo a equação 1. A capacidade de intermediação de um vértice é o número de caminhos mínimos que passam por esse vértice.*

Essa definição é similar a definição de *Betweenness Centrality*, uma métrica de

Redes Complexas[2].

1.2. Implementação

A linguagem escolhida para a implementação do código foi a linguagem *Python* versão 3.5, que se encontra disponível nas máquinas do *DCC* através do comando `python3.5`. As bibliotecas utilizadas foram `heapq` que define uma estrutura de heap (fila de prioridades), `copy` para clonar objetos e `operator` para obter o *index* de vetores. **TODAS** as bibliotecas utilizadas são *built-in* da linguagem *Python*.

Uma vez que *Python* possui suporte a Programação Orientada a Objetos, as estruturas de dados foram modularizadas em classes e essas classes agrupadas em módulos. Na raiz do trabalho se encontra o módulo `political_problem` que contém todas as estruturas e algoritmos utilizados e o módulo `main` que utiliza a interface fornecida pelo anterior para resolver o problema. A explicação dos módulos será apresentada a seguir:

1.2.1. Módulo `node.py`

Contém a classe `Node` que representa um nó de um grafo, possui um identificador e os nós vizinhos armazenados em uma estrutura de **Lista de Adjacências** (que pode ser facilmente representada por um *dictionary* em *Python*). Possui suporte a obtenção dos identificadores, vizinhos e à aplicação da normalização.

1.2.2. Módulo `graph.py`

Contém a classe `Graph` que representa um grafo, possui um dicionário de nós identificados pelos identificadores (como os próprios nós armazenam seus vizinhos, `Graph` não precisa armazenar essa informação novamente). Possui suporte a cópia, aplicação de função de normalização em arestas e cálculo de distâncias.

1.2.3. Módulo `graph.utils.py`

Contém funções para leitura da entrada e *parsing* dos dados. Dado um arquivo corretamente formatado, retorna um objeto `Graph` (Figura 2).

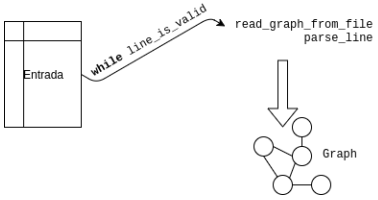

Figura 2. Esquema de leitura e *parsing* dos arquivos. As três funções do módulo agem através da função principal `read_graph_from_file` e geram o Grafo corretamente.

[2] `https://en.wikipedia.org/wiki/Betweenness_centrality`

1.2.4. Módulo `political_problems.utils.py`

Contém as funções responsáveis por escrever a resposta em arquivo. A função `print_output` recebe os somatórios do *betweenness* de cada vértice e os imprime na ordem correta. Para isso é feita uma ordenação por id. Depois é feita uma outra ordenação em ordem decrescente pelo valor do *betweenness*. Essa ordenação é feita utilizando um **algoritmo estável**, dessa forma, é seguida a regra do critério de desempate.

1.2.5. Módulo `algorithms.py`

Contém a implementação dos principais algoritmos utilizados no trabalho, sendo eles:

- **Normalização de arestas**: Dado um grafo G, cria um grafo G' cujas arestas foram normalizadas segundo a equação 1
- **Caminho Mínimo**: Uma implementação do algoritmo de *Dijkstra* (Caminhos mínimos com única fonte) para o caminho mínimo entre um vértice s para todos os vértices de um grafo. O algoritmo é implementado na função `dijkstra_algorithm` que retorna um dicionário contendo os identificadores de cada nó e seu predecessor. (Figura 3).
- **Cálculo do *Betweenness*:** Utiliza uma adaptação do algoritmo de *Johnson* (Caminhos mínimos com múltiplas fontes) para calcular o número de caminhos mínimos que passam por cada vértice. Note que não é necessário realizar a etapa de transformar as arestas negativas, pois os valores originais das arestas estão no intervalo [0,100]. As funções `calculate_betweenness` e `update_betweenness_distances` executam o *Dijkstra* para cada vértice e armazenam o somatório do número de caminhos mínimos que passam por cada vértice.

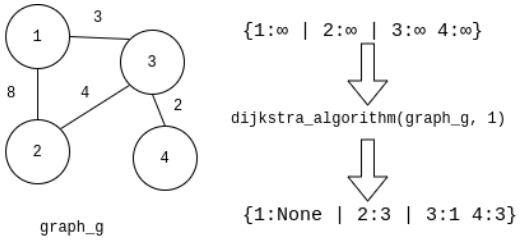

Figura 3. Esquema do funcionamento da implementação do algoritmo de *Dijkstra*, o retorno da função é um dicionario para poder ser utilizado pelo algoritmo que calcula o *Betweenness*

.

1.2.6. Módulo `graph_heap.py`

Contém a classe `GraphHeap`, uma fila de prioridades utilizada pelo algoritmo de *Dijkstra*. Para isso, a mesma faz uso da estrutura `heapq`, que mantém a estrutura de *heap* dentro de uma *list*. Para fins práticos, a lista tambem armazena os pesos dos vértices que já não estão mais contidos na mesma.

1.2.7. Módulo `main.py`

O módulo *main* é responsável por chamar cada um dos algoritmos e funções na ordem correta e por fim, gravar a resposta em arquivo.

2. Cálculo de Complexidade

Para calcular a complexidade da solução, utilizaremos a letra v para representar o número de vértices e a letra e para representar o número de arestas. As análises serão divididas por módulo.

2.1. Módulo `node.py`

Função	Complexidade de Tempo	Complexidade de Espaço
`Node.add_neighbors`	$\theta(1)$	$\theta(e)$
`Node.add_neighbor`	$\theta(1)$	$\theta(1)$
`Node.get_neighbors_as_list`	$\theta(e)$	$\theta(1)$
`Node.normalize_weights`	$\theta(1)$	$\theta(1)$
`Node.get_weight`	$\theta(1)$	$\theta(1)$

2.2. Módulo `graph.py`

Função	Complexidade de Tempo	Complexidade de Espaço
`add_node`	$\theta(1)$	$\theta(e)$
`link_nodes`	$\theta(1)$	$\theta(1)$
`normalize_graph_weights`	$\theta(e)$	$\theta(1)$
`get_nodes`	$\theta(v)$	$\theta(v)$
`duplicate`	$\theta(v+e)$	$\theta(v+e)$
`get_node_neighbors`	$\theta(e)$	$\theta(e)$
`get_distance_between`	$\theta(1)$	$\theta(1)$

2.3. Módulo `graph.utils.py`

Função	Complexidade de Tempo	Complexidade de Espaço
`read_graph_from_file`	$\theta(e)$	$\theta(v+e)$
`line_is_not_valid`	$\theta(1)$	$\theta(1)$
`parse_line`	$\theta(1)$	$\theta(1)$

2.4. Módulo `political_problems.utils.py`

Função	Complexidade de Tempo	Complexidade de Espaço
`print_output`	$\theta(1)$	$\theta(1)$
`sort_vertex_by_id`	$\theta(vlog(v))$	$\theta(1)$
`sort_vertex_by_id`	$\theta(vlog(v))$	$\theta(1)$

2.5. Módulo `graph_heap.py`

Função	Complexidade de Tempo	Complexidade de Espaço
`GraphHeap.push_vertex`	$\theta(log(v))$	$\theta(1)$
`GraphHeap.update_vertex`	$\theta(log(v))$	$\theta(1)$
`GraphHeap.pop_vertex`	$\theta(1)$	$\theta(1)$
`GraphHeap.is_empty`	$\theta(1)$	$\theta(1)$
`GraphHeap.get_distance`	$\theta(v)$	$\theta(1)$

2.6. Módulo `algorithms.py`

Função	Complexidade de Tempo	Complexidade de Espaço
`normalize_graph`	$\theta(ve)$	$\theta(v+e)$
`shortest_path`	$\theta((v+e)log(v))$	$\theta(v)$
`dijkstra_algorithm`	$\theta((v+e)log(v))$	$\theta(v)$
`calculate_betweenness`	$\theta((v^2+ve)log(v))$	$\theta(v)$
`update_betweenness_distances`	$\theta(v)$	$\theta(1)$

2.7. Complexidade Final

As operações dentro de `node.py` e `graph.py` possuem custos baixos por serem operações simples em grafos. A implementação com listas de adjacências faz com que sejam no máximo lineares em vértices ou arestas. O mesmo raciocínio pode ser aplicado ao módulo `graph_heap.py`, que se trata de uma implementação de um *binary heap* com suporte a *update* de peso.

Ambos os módulos de utilidades possuem funções para tratamento de entrada e saída, onde a leitura de entrada é linear no número de arestas e a saída possui complexidade $\theta(vlog(v))$ devido a ordenação (utilizando o método *built-in* do *Python*).

Portanto, a complexidade final é limitada pelo próprio algoritmo de calcular o *betweenness* roda um algoritmo de *Dijkstra* para cada vértice, portanto a complexidade de tempo final do programa é:

$$\theta((v^2+ve)log(v))$$

A complexidade de espaço se mantém linear $\theta(v+e)$, dado que as estruturas utilizadas são bem simples (como listas e dicionários).

3. Experimentação

A implementação do problema executa com sucesso todas as entradas disponibilizadas para testes. Mas uma vez que a complexidade de tempo não é trivial, é desejável desenvolver testes para medir o tempo de execução do trabalho em relação ao aumento de uma das entradas.

A máquina onde os testes foram realizados possui as seguintes especificações: 16GB de RAM DDR3, com processador *Intel Core i5-7400 CPU @ 3.00GHz x4* e *SSD* de 512GB. Sistema Operacional: *GNU/Linux Ubuntu 16.04 x64*.

3.1. Aumentando o número de vértices

Para o primeiro teste, aumenta-se o número de vértices em cada execução. A chance de dois vértices possuírem uma aresta respeita a probabilidade em um distribuição uniforme de 10%. A figura 4 mostra a evolução ao longo do tempo.

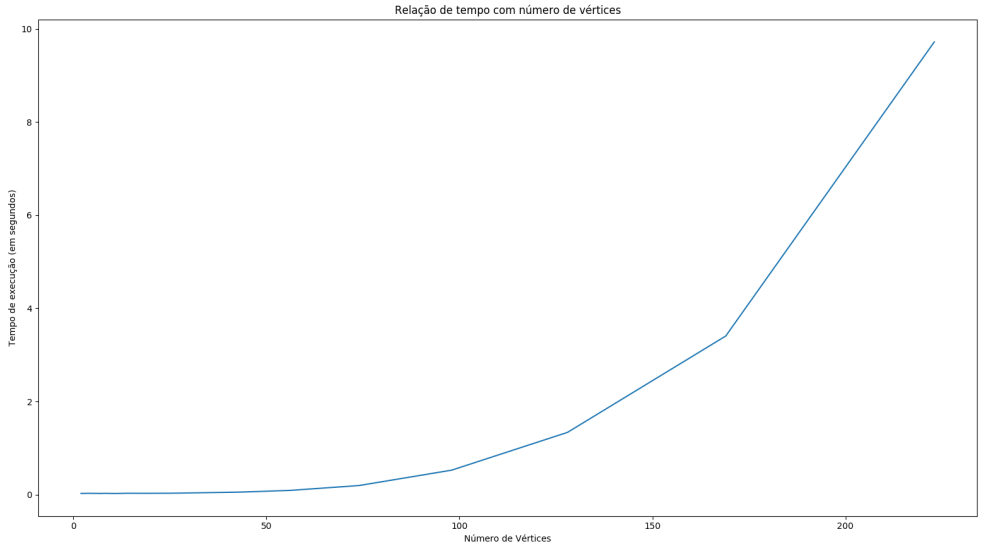

Figura 4. Evolução do tempo de execução do programa a medida que se aumenta o número de vértices. É possível notar que o tempo é muito penalizado com o aumento do número de vértices. Esse comportamento era esperado, uma vez que é executado o algoritmo de *Dijkstra* para cada vértice, a complexidade é multiplicada por um fator dependente de v. (Vale lembrar que o crescimento acompanha o quadrado de v multiplicado pelo logaritmo do mesmo).

3.2. Aumentando o número de arestas

Para esse teste, o número de vértices foi fixado em 150 e os testes modificam a probabilidade dos vértices possuírem uma aresta. A figura 5 mostra o gráfico relativo ao teste.

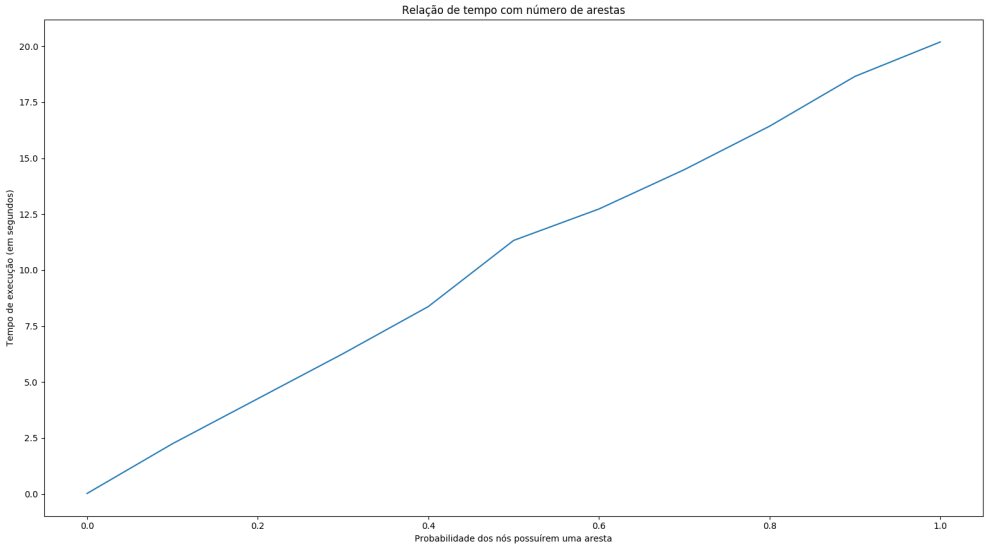

Figura 5. Evolução do tempo de execução do programa a medida que se aumenta o número de arestas. Note que o tempo aumenta quase linearmente, isso se deve ao fato de que a ordem de complexidade do programa é influenciada pelo número de arestas de forma linear. (Observando a notação assintótica: $\theta((v^2 + ve)log(v))$ é possível ver que o termo e referente ao número de arestas aumenta a complexidade linearmente.)